Machine Learning

Title of the project: Optical Interconnection Network

Akash Maurya, akashmaury14@gmail.com

ML Category: Regression

---

1. Introduction

An Optical Interconnection Network (OIN) is a type of communication network that uses optical technology to transmit data signals between different electronic components or systems. The OIN technology typically includes a set of optical fibres, waveguides, connectors, and other optical devices to facilitate data transmission.

OINs are used in a variety of applications such as data centres, high-performance computing, and telecommunication systems, where the high bandwidth and low latency provided by optical communication are critical for efficient data transfer. In these applications, OINs offer several advantages over traditional electronic interconnects, including higher data rates, lower power consumption, and reduced signal distortion.

Overall, OINs are an important technology for enabling high-speed, low-latency communication between electronic components and systems in a variety of applications, making them an essential component of modern computing and communication infrastructure.

"Optimising the Performance of an Optical Interconnection Network Using Machine Learning: A Study of the Impact of Node and Thread Numbers, Spatial and Temporal Distributions, Channel Utilisation, Message Transfer time and threat run time, Channel Waiting time, Input waiting time on the Processor Utilisation ."

2. Dataset and Features

The "Optical Interconnection Network Dataset" has 10 attributes and 640 number of instances of performance measurement from a simulation of 2-Dimensional Multiprocessor Optical Interconnection Network.

Hereunder is the brief description about all the attributes/features of the dataset:

Node Number: Refers to the quantity of nodes within the network, which can either be 8x8 or 4x4.

Thread Number: Indicates the initial quantity of threads in each node during the simulation.

Spatial Distribution: evaluates network performance by using synthetic traffic workloads. It comprises four traffic models, which are Uniform (UN), Hot Region (HR), Bit reverse (BR), and Perfect Shuffle (PS).

Temporal Distribution: Involves the temporal distribution of packet generation. It is executed by independent traffic sources, including client-server traffic and asynchronous traffic.

T/R: Denotes the message transfer time (T) and thread run time (R) in the simulation. T is uniformly distributed with a mean that falls within the range of 20 to 100 clock cycles, while R is exponentially distributed, with an average of 100 clock cycles.

Channel Waiting Time: Refers to the average waiting time of a packet in the output channel queue before the channel services it.

Input Waiting Time: Represents the average waiting time of a packet before the processor services it.

Network Response Time: The duration between enqueuing a request message at the output channel and receiving the corresponding data message in the input queue.

Channel Utilization: Measures the percentage of time that the channel spends transferring packets within the network.

Processor Utilization: Calculates the average processor utilization, which is the percentage of time that threads run on the processor.

Succeeding are the steps we applied for the exploratory data analysis of the dataset:

Before reading the csv file, we found that the values were separated with semicolons(';') instead of commas(',') and had commas(',') instead of decimals('.') in the records.

We used the following methods to read the csv file and used ";" as the separating parameter.

```
df_data=
pd.read_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/
00449/optical_interconnection_network.csv",sep=";" or ",")
df_data
```

We replaced the ',' with the '.' using the following code:

```
df_data=df_data.replace(",",".", regex=True)
```

The data set had 5 unwanted columns with NaN values, so we used the following syntax to drop them from the data set:

```
df_data=df_data.drop(['Unnamed: 10','Unnamed: 11','Unnamed:
12','Unnamed: 13','Unnamed: 14'],axis=1)
```

Using the .info() function we got to know that the numerical features have the data type as "object". We converted the datatypes of these features from object to float datatype using the following looping code:

```
for i in df_data.columns:
    df_data[i]=df_data[i].apply(pd.to_numeric,errors='ignore')
```

```
df_data.info()
✓ 0.1s
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 640 entries, 0 to 639
Data columns (total 10 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   Node Number            640 non-null    int64
 1   Thread Number          640 non-null    int64
 2   Spatial Distribution   640 non-null    object
 3   Temporal Distribution  640 non-null    object
 4   T/R                    640 non-null    object
 5   Processor Utilization  640 non-null    object
 6   Channel Waiting Time   640 non-null    object
 7   Input Waiting Time     640 non-null    object
 8   Network Response Time  640 non-null    object
 9   Channel Utilization    640 non-null    object
dtypes: int64(2), object(8)
memory usage: 50.1+ KB
```

Now, we are done with removing the errors and the preprocessing of the dataset and it is ready to be used for the analysis.

We used the following syntax to get the basic information like range index, tonal non-null values, data types, columns and count of each type of data type columns.
The data set has the range index from 0 to 639 counting for 640 records, with no null values and 6 float64 type, 2 int64 type and 2 object type columns.
df_data.info()

```
df_data.info()
✓ 0.0s
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 640 entries, 0 to 639
Data columns (total 10 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   Node Number            640 non-null    int64
 1   Thread Number          640 non-null    int64
 2   Spatial Distribution   640 non-null    object
 3   Temporal Distribution  640 non-null    object
 4   T/R                    640 non-null    float64
 5   Processor Utilization  640 non-null    float64
 6   Channel Waiting Time   640 non-null    float64
 7   Input Waiting Time     640 non-null    float64
 8   Network Response Time  640 non-null    float64
 9   Channel Utilization    640 non-null    float64
dtypes: float64(6), int64(2), object(2)
memory usage: 50.1+ KB
```

Used .describe() function to get the statistical analysis of the data. The mean of our target variable i.e "Processor Utilization" is 0.649013 and standard deviation of 0.194737 and the values range from min-0.202377 to max-0.986516.

```
df_data.describe().T
✓ 1.3s
```

	count	mean	std	min	25%	50%	75%	max
Node Number	640.0	40.000000	24.018772	16.000000	16.000000	40.000000	64.000000	64.000000
Thread Number	640.0	7.000000	2.237817	4.000000	5.500000	7.000000	8.500000	10.000000
Spatial Distribution	640.0	1.500000	1.118908	0.000000	0.750000	1.500000	2.250000	3.000000
Temporal Distribution	640.0	0.500000	0.500391	0.000000	0.000000	0.500000	1.000000	1.000000
T/R	640.0	0.550000	0.287453	0.100000	0.300000	0.550000	0.800000	1.000000
Processor Utilization	640.0	0.649013	0.194737	0.202377	0.492530	0.624787	0.833106	0.986516
Channel Waiting Time	640.0	377.459157	381.974899	0.950721	29.247560	265.614624	664.965408	1627.330246
Input Waiting Time	640.0	333.247102	233.721860	33.036130	137.730986	261.855556	485.943680	892.852416
Network Response Time	640.0	1504.247529	1202.606968	0.529210	580.676198	1232.150369	2115.326618	6065.736672
Channel Utilization	640.0	26.347886	223.782214	0.136979	0.587539	0.773611	0.905573	2895.323131

We calculated the correlation between each feature and the relation of the independent columns with the dependent one. We can infer from the absolute values that the 'Input Waiting Time', 'Channel Waiting Time' and 'Network Response Time' have the maximum correlation with the target column 'Processor Utilisation".

```
df_data.corr()
```
✓ 0.1s Python

```
C:\Users\HP\AppData\Local\Temp\ipykernel_8160\3492733786.py:1: FutureWarning: The default value of numeric_only in DataFrame.corr is deprecated. In a future
  df_data.corr()
```

	Node Number	Thread Number	T/R	Processor Utilization	Channel Waiting Time	Input Waiting Time	Network Response Time	Channel Utilization
Node Number	1.000000e+00	-8.395103e-16	3.221084e-18	-0.068037	-0.012737	0.028482	0.144028	0.114445
Thread Number	-8.395103e-16	1.000000e+00	-4.321537e-18	0.234490	0.281104	0.508769	0.341949	-0.153666
T/R	3.221084e-18	-4.321537e-18	1.000000e+00	-0.660957	0.873277	-0.769204	0.631521	0.051454
Processor Utilization	-6.803708e-02	2.344896e-01	-6.609570e-01	1.000000	-0.531863	0.579342	-0.342024	-0.046999
Channel Waiting Time	-1.273671e-02	2.811040e-01	8.732773e-01	-0.531863	1.000000	-0.639091	0.713670	-0.030036
Input Waiting Time	2.848191e-02	5.087694e-01	-7.692040e-01	0.579342	-0.639091	1.000000	-0.378257	-0.093908
Network Response Time	1.440280e-01	3.419492e-01	6.315207e-01	-0.342024	0.713670	-0.378257	1.000000	-0.142961
Channel Utilization	1.144453e-01	-1.536656e-01	5.145444e-02	-0.046999	-0.030036	-0.093908	-0.142961	1.000000

Here is the visualisation for the complete data set in 3D using PCA and tSNE . Here is the code and the visualisation for both the techniques.

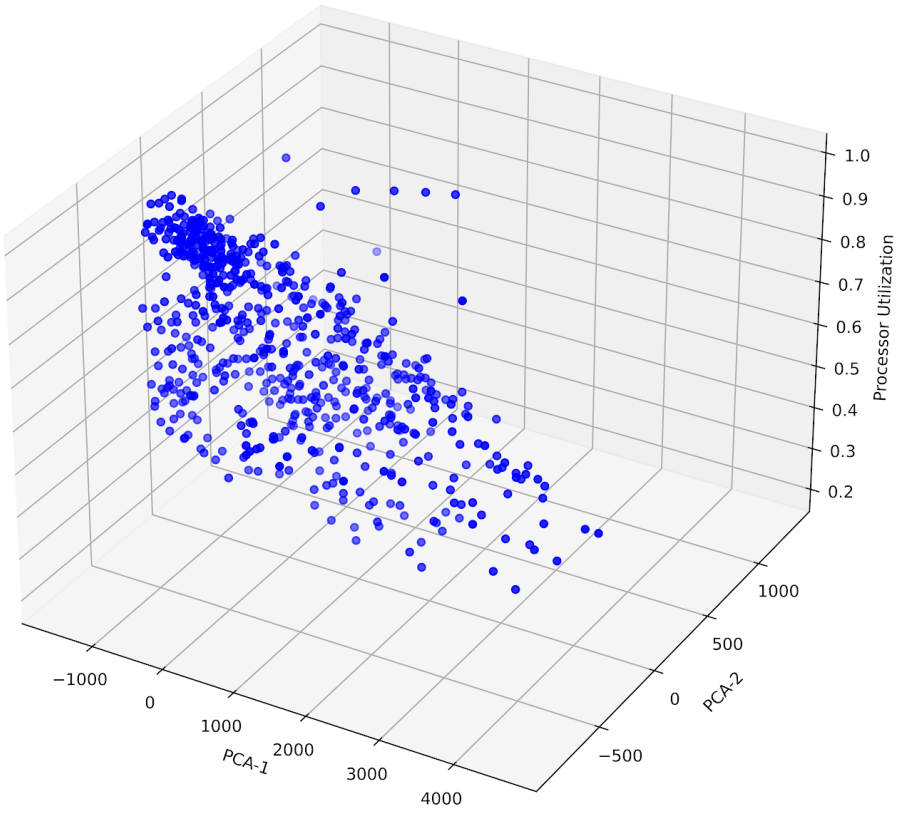
Principal Component Analysis (PCA) ;

> for 2D conversion using PCA

```
from sklearn.decomposition import PCA
pca = PCA(n_components= 2)

pca.fit(X1)
```

> plotting the 3D graph

```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(15, 10))
ax = fig.add_subplot(projection='3d')

ax.scatter(X1_2D[:,0], X1_2D[:,1], y1, color="b");
ax.set_xlabel("PCA-1")
ax.set_ylabel("PCA-2")
ax.set_zlabel("Processor Utilization")

plt.tight_layout()
plt.show()
```



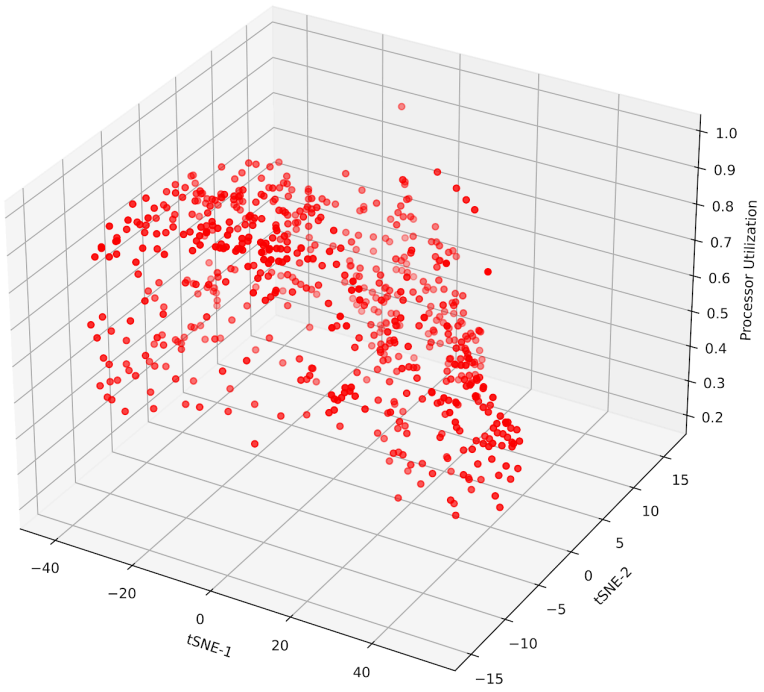
tSNE Dimensionality reduction:

>for 2D conversion using tSNE

```
from sklearn.manifold import TSNE
tSNE = TSNE(n_components= 2,init='pca',learning_rate='auto')

pca.fit(X1)
```

>plotting the 3D graph:

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(figsize=(15, 10))
ax = fig.add_subplot(projection='3d')

ax.scatter(X1_2D_tsne[:,0], X1_2D_tsne[:,1], y1, color="r");
ax.set_xlabel("tSNE-1")
ax.set_ylabel("tSNE-2")
ax.set_zlabel("Processor Utilization")

plt.tight_layout()
plt.show()
```



3. Methods

Regression Model:

Regression method is a supervised machine learning model used to predict continuous values. Regression is one of the most powerful models for making these kinds of models.

It takes the inputs, transforms the input into model parameters and calculates the output using these parameters.

3.1 Baseline - Linear Regression

The baseline method used here for the modelling is the "Multivariate Linear Regression". Multivariate linear regression is a statistical technique used to model the relationship between multiple independent variables and a dependent variable. In multivariate linear regression, the goal is to find a linear equation that best represents the relationship between the independent variables and the dependent variable.

The equation takes the form:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \ldots\ldots\ldots\ldots + \theta_n x_n$$

where 👍

- $\hat{y}$- is the predicted values
- $x_1, x_2, x_3 \ldots\ldots x_n$ are the independent variables
- θ_0 is the intercept of the equation
- $\theta_1, \theta_2, \theta_3, \ldots\ldots \theta_n$ are the coefficients of the independent variables

The cost function for this model is the "Mean Squared Error". The equation is:

$$\text{MSE}(X, h_0) = \frac{1}{m}\sum_{i=1}^{m}(\theta^T X^{(i)} - y^{(i)})^2$$

To find the θ parameters we have a closed form solution - 'Normal Equation'. The equation is:

$$\hat{\theta} = (X^T \cdot X)^{-1} \cdot (X^T \cdot y)$$

$\hat{\theta}$ are the values of the θ that minimises the cost function.

We have used the class/methods from the scikit-learn library to train the model and also to generate the predictions. Following is the code for training the data using the class called 'LinearRegression':

```
from sklearn.linear_model import LinearRegression
model=LinearRegression()
model.fit(X_train,y_train)
```

Mentioned below is the method used to predict the value of target variable:

```
model.predict(X_test)
```

3.2 Polynomial Regression

Polynomial regression is a statistical method used to model nonlinear relationships between a dependent variable and one or more independent variables. Unlike linear regression, which assumes a linear relationship between the variables, polynomial regression models can capture more complex, curvilinear relationships.

In polynomial regression, a polynomial equation is fitted to the data that relates the dependent variable to the independent variable(s).

The equation takes the form 👍

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x^2_2 + \theta_3 x^3_3 + \ldots\ldots\ldots + \theta_n x^n_n$$

where 👍

- $\hat{y}$- is the predicted values
- $x_1, x_2, x_3 \ldots\ldots x_n$ are the independent variables
- θ_0 is the intercept of the equation
- $\theta_1, \theta_2, \theta_3, \ldots\ldots \theta_n$ are the coefficients of the independent variables

3.3 Regularization

Regularization is a technique used in machine learning and statistical modelling to prevent overfitting and improve the generalisation performance of a model. Overfitting occurs when a model learns

the noise in the training data rather than the underlying pattern, which leads to poor performance on new, unseen data.

Regularization works by adding a penalty term to the loss function that the model optimises during training. The penalty term discourages the model from fitting the noise in the data by constraining the weights of the model.

Mainly there are two type of Regularization 👍

➡️Lasso (L1) Regularization:

In L1 regularization, the penalty term is the sum of the absolute values of the weights. This encourages the model to set some weights to zero, effectively performing feature selection and simplifying the model.

The equation is like 👍

$$J(\theta) = MSE(\theta) + 2\alpha \sum_{i=1}^{n} abs(\theta_i)$$

➡️Ridge (L2) Regularization:

In L2 regularization, the penalty term is the sum of the squares of the weights. This encourages the model to distribute the weight evenly across all the features, effectively reducing the impact of any individual feature on the outcome.

The equation is like 👍

$$J(\theta) = MSE(\theta) + \frac{\alpha}{m} \sum_{i=1}^{n} abs(\theta_i^2)$$

➡️ Elastic Net Regularization:

It is the combination of L1 and L2 regularization.

The equation is like 👍

$$J(\theta) = MSE(\theta) + r\left(2\alpha \sum_{i=1}^{n} abs(\theta_i)\right) + (1 - r)\left(\frac{\alpha}{m} \sum_{i=1}^{n} abs(\theta_i^2)\right)$$

3.4 Support Vector Machine(SVM) Regression

Support Vector Machine (SVM) Regression is a machine learning algorithm that is used for regression analysis. It is a variant of the SVM algorithm, which is primarily used for classification problems.

The goal of SVM Regression is to find a function that can best fit a given set of data points while minimising the error between the predicted and actual values. The algorithm works by finding a hyperplane that maximises the margin (i.e., the distance between the hyperplane and the closest data points) while ensuring that a certain percentage of the data points lie within a specified margin. The hyperplane is determined by a subset of the training data, called support vectors, which lie closest to the hyperplane.

SVM can handle both linear and polynomial regression problems. It uses various techniques to handle these problems :

➡️Linear SVM Regression:

The objective of Linear SVM Regression is to find a linear function that can best fit a given set of data points while minimising the error between the predicted and actual values. It is able to handle

high-dimensional data, resistance to overfitting, and has the ability to model linear relationships between variables.

➡️Polynomial Kernel SVM Regression:

The objective of Polynomial Kernel SVM Regression is to find a non-linear function that can best fit a given set of data points while minimising the error between the predicted and actual values. The algorithm works by transforming the input variables into a higher-dimensional feature space using a polynomial kernel function.

The polynomial kernel function maps the input variables into a higher-dimensional feature space with a linear hyperplane that can fit the data points. The degree of the polynomial kernel function determines the complexity of the non-linear function

➡️Radial Basis Function(RBF) Kernel SVM Regression:

The objective of RBF Kernel Regression is to find a non-linear function that can best fit a given set of data points while minimising the error between the predicted and actual values. The algorithm works by transforming the input variables into a higher-dimensional feature space using an RBF kernel function.

The RBF kernel function maps the input variables into a higher-dimensional feature space where a non-linear function can fit the data points. The RBF kernel function uses a parameter called gamma that determines the width of the kernel function. A smaller value of gamma results in a wider kernel function, which makes the function smoother, while a larger value of gamma results in a narrower kernel function, which makes the function more complex.

3.5 Decision Tree

Decision trees for regression, also known as regression trees, are machine learning models used for predicting continuous numerical values. They work by recursively splitting the input data based on different features to create a tree-like structure of decision nodes and leaf nodes. Each internal node represents a feature and a splitting criterion, while each leaf node contains a predicted value.

During training, the decision tree algorithm selects the best feature and splitting criterion at each node based on a measure of impurity reduction, such as mean squared error or variance. This process continues until a stopping criterion is met, such as a maximum depth or a minimum number of samples per leaf.

➡️Ensemble Learning

Ensemble learning for regression is a technique that combines multiple individual regression models to improve prediction accuracy and robustness. It works by training a group or ensemble of regression models and aggregating their predictions to obtain a final prediction. It finds the mean of all the predicted values that are predicted by the assembled models and gives as the final result for the input data.

3.6 Random Forest Regressor

Random Forest for regression is an ensemble learning method that combines multiple decision trees to make accurate predictions for continuous numerical values. It works by creating a random subset of features and constructing an ensemble of decision trees, known as a forest. Each tree is trained on a different subset of the training data and uses a random subset of features for splitting at each node.

During prediction, the random forest aggregates the predictions from all the individual trees to produce a final prediction. This aggregation is done by averaging the predictions and giving the final prediction.

3.7 Adaptive Boosting Regressor (AdaBoost)

AdaBoost Regressor is a popular ensemble learning algorithm in machine learning. It combines multiple weak regression models, typically decision trees, to create a strong predictive model. The algorithm assigns weights to each data point, with higher weights given to the points that were poorly predicted by previous models. It iteratively trains weak models on reweighted data, where the weights are adjusted based on the errors of the previous models. The final prediction is obtained by aggregating the predictions of all weak models, with each model's contribution weighted by its performance

3.8 GradientBoost Regressor

GradientBoost Regressor is a powerful machine learning algorithm that builds an ensemble of weak regression models in a sequential manner. It starts by fitting an initial model to the data and then subsequent models are trained to correct the errors made by the previous models. Each model is trained to minimise the residual errors of the ensemble. The predictions of all models are combined, with each model's contribution weighted by a learning rate. This iterative process continues until a specified number of models are created or a convergence criterion is met. It is known for its ability to handle complex relationships and produce highly accurate predictions in regression tasks.

3.9 Hyperparameter Tuning

Hyperparameters are settings or configurations that are not learned from the data but are set by the user before training the model.

Hyperparameter tuning is the process of finding the optimal set of hyperparameters for a machine learning algorithm to maximise its performance.

Two commonly used techniques for hyperparameter tuning are GridSearchCV and RandomSearchCV:

GridSearchCV:

GridSearchCV exhaustively searches through a predefined grid of hyperparameter combinations. It evaluates the performance of the model using cross-validation for each combination and selects the one that yields the best performance.

RandomSearchCV:

RandomSearchCV randomly samples from the hyperparameter space. It allows the user to define a distribution or range for each hyperparameter and the number of iterations to perform. It selects random combinations of hyperparameters and evaluates the performance using cross-validation. This technique is beneficial when the hyperparameter space is large, as it can efficiently explore a wide range of values.

4. Experiments & Results

4.1 Protocol

We have split the entire dataset in the 80% training and 20% testing dataset. We used the 'train_test_split' method to divide independent and dependent variables into training and testing dataset. We have seeded the value of the random state to 42, so that the results are reproducible and we are able to get the same random splitting everytime we run the program.
Following is the code for the same:

```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test=train_test_split(X, y, test_size=0.2,
                                                random_state=42)
```

We converted these dataframes as a np-array, because the 'sklearn' library works better with the arrays. Code for the conversion is 👍

```
type(X_train)
X_train,X_test,y_train,y_test= X_train.values, X_test.values,
y_train.values, y_test.values
```

We have trained three baseline models using various feature selections and feature scaling methods.

Our first model **'model'** is trained on the feature selected dataset. The selected features are the 'Node Number', 'Thread Number', 'Input Waiting Time', 'Network Response Time' and 'Channel Utilization'. We have dropped out the features with the string entries.

The second model is **'model1'**. We have randomly allocated the numerical values to the categorical string values in the features and then trained this model. We did the following transformations in the features:

```
df_data["Spatial Distribution"]=df_data["Spatial
Distribution"].replace(['UN','HR', 'BR', 'PS'],[0,1,2,3])

df_data["Temporal Distribution"]=df_data["Temporal
Distribution"].replace(['Client-Server','Asynchronous'],[0,1])
```

The third model **'model2'** is trained using the feature scaling. We have standardized the training data set and trained the model using the standardized dataset. Used the following code for the standardization:

```
from sklearn.preprocessing import StandardScaler
sc= StandardScaler()
sc.fit(X1_train)

X1_train_std=sc.transform(X1_train)
X1_test_std=sc.transform(X1_test)
```

We have the **polynomial model**, trained on the polynomial transformation of the standardized training data set. We used the following code to transform the data to 2 degree polynomial and to train the polynomial model:

```
from sklearn.preprocessing import PolynomialFeatures
poly_features= PolynomialFeatures(degree=2, include_bias=False)

X_train_poly=poly_features.fit_transform(X1_train_std)
X_test_poly = poly_features.fit_transform(X1_test_std)
model_poly = LinearRegression()
model_poly.fit(X_train_poly,y1_train)
```

We have tried out three different types of regularization on the polynomial data set and prepared the models accordingly.

➡️L1 Regularization (LASSO):

We have used the following code to implement this regularization:

```
from sklearn.linear_model import Lasso

model_lasso = Lasso(alpha=0.0005)
model_lasso.fit(X_train_poly, y1_train)
```

➡️L2 Regularization (Ridge):

We have used the following code to implement this regularization:

```
from sklearn.linear_model import Ridge

model_ridge = Ridge(alpha=10, solver="cholesky")
model_ridge.fit(X_train_poly, y1_train)
```

➡️Elastic Net Regularization:

It is the combination of L1 and L2 regularization. We used the following data set to implement this regularization:

```
from sklearn.linear_model import ElasticNet

model_elastic_net = ElasticNet(alpha=0.0005, l1_ratio=1)
model_elastic_net.fit(X_train_poly, y1_train)
```

We have tried “Support Vector Machine” for Regression. We have used it with different kernel approaches and developed the models using the standarized training data set. We have developed three models with different kernels.

The code for importing the support vector regression from the support vector machine.

```
from sklearn.svm import SVR
```

➡️Linear Kernel SVM Model:

We used the linear kernel to get the prediction model. Following is the code used to develop this model:

```
svm_li = SVR(kernel='linear', C=1)

svm_li.fit(X1_train_std,y1_train)
```

➡️Polynomial Kernel SVM Model:

We used the polynomial kernel to get the prediction model. Following is the code used to develop this model:

```
svm_poly = SVR(kernel ='poly', C=1)

svm_poly.fit(X1_train_std,y1_train)
```

➡️RBF Kernel SVM Model:

We used the Radial Basis Function(RBF) kernel to develop the prediction model. Following is the code used to develop this model:

```
svm_rbf = SVR(kernel='rbf', C=1, gamma=0.5)

svm_rbf.fit(X1_train_std,y1_train)
```

➡️Decision Tree Model:

We used the decision tree to develop the prediction model. Following is the code used to develop this model:

```
from sklearn.tree import DecisionTreeRegressor
tree_reg =DecisionTreeRegressor(max_depth=12)
tree_reg.fit(X1_train_std,y1_train)
```

➡️Ensemble Learning:

👉 Random Forest:

We used the random forest method of ensemble learning to develop the prediction model. Following is the code for the this model:

```
from sklearn.ensemble import RandomForestRegressor
rnd_forest = RandomForestRegressor(n_estimators=500,max_leaf_nodes=15,
                                                              n_jobs=-1)
rnd_forest.fit(X1_train_std,y1_train)
```

👉 AdaBoostRegressor:

We used the AdaBoost method of ensemble learning to develop the prediction model. Following is the code for the this model:

```
from sklearn.ensemble import AdaBoostRegressor
ada_boost = AdaBoostRegressor(DecisionTreeRegressor(max_depth=10),
n_estimators=500,learning_rate=0.5)

ada_boost.fit(X1_train_std,y1_train)
```

👉 GradientBoostRegressor:

We used the gradient boost method of ensemble learning to develop the prediction model. Following is the code for the this model:

```
from sklearn.ensemble import GradientBoostingRegressor
grd_boost = GradientBoostingRegressor(max_depth=15, n_estimators=500,
learning_rate=1)

grd_boost.fit(X1_train_std,y1_train)
```

➡️Hyperparameter Tuning:

We have used the hyperparameter tuning to get the best hyperparameters for the models. Hear is the code for the two types of hyperparameter tuning:

For GridSearchCV:

```
hpt_rnd_forest = RandomForestRegressor()

from sklearn.model_selection import GridSearchCV

learnRate = [0.2, 0.3, 0.4, 0.5]
```

```
hpt_param_grid = [
    {'n_estimators':[i for i in range(10,100,10)],
     'max_depth': [j for j in range(2,25,2)]}
    ]

print(hpt_param_grid)
```

```
grid_search_rnd_forest = GridSearchCV(estimator = hpt_rnd_forest,
                            param_grid = hpt_param_grid,
                            cv = 5)

grid_search_rnd_forest.fit(X1_train_std,y1_train)
```

```
print(grid_search_rnd_forest.best_params_)

print(grid_search_rnd_forest.best_estimator_)

print(grid_search_rnd_forest.best_score_)
```

For RandomSearchCV:

```
hpt_rnd_forest = RandomForestRegressor()
from sklearn.model_selection import RandomizedSearchCV
```

```
hpt_param_dist = {'n_estimators':[i for i in range(10,101,10)],
                  'max_depth': [j for j in range(2,25,2)]}

hpt_param_dist
random_search_rnd_forest = RandomizedSearchCV(estimator =
hpt_rnd_forest, param_distributions=hpt_param_dist, cv=5)

random_search_rnd_forest.fit(X1_train_std,y1_train)
```

```
print(random_search_rnd_forest.best_params_)

print(random_search_rnd_forest.best_estimator_)

print(random_search_rnd_forest.best_score_)
```

4.2 Results

We have evaluated the accuracy of our models as the R^2 value and the cross validation score.

R^2**(Coefficient of determination)** - is the comparison of how good is your model as compared to the simplest model i.e. mean value model. Equation is as follows:

$$R^2 = 1 - \frac{Squared\ Sum_{(res)}}{Squared\ Sum_{(total)}}$$

Cross Validation - It gives how robust the model is to the small changes in the training and testing dataset.

The final score is presented as the mean of the R^2 values calculated in the cross validation and the standard deviation of these values.
Following is the code for R^2 and 10 fold cross validation 👍

```
from sklearn.model_selection import cross_val_score
Cross_val_R2=cross_val_score(model,X_train,y_train,cv=10)
print(Cross_val_R2)

#mean and standard deviation
print(Cross_val_R2.mean())
print(Cross_val_R2.std())
```

So the cross validation score of the three models are as follows 👍

Linear Regression Baseline models:

Model-01 (Name -'model'):

Cross Validation Score : $0.353 + / - 0.092$

Model-02 (Name - 'model1'):

Cross Validation Score : $0.671 + / - 0.056$

Model-03 (Name - 'model2'):

Cross Validation Score : $0.671 + / - 0.056$

Polynomial Regression Model:

Model-01 (Name - 'model_poly'):

Cross Validation Score : $0.911 + / - 0.038$

Regularization Models:

Model-01 (Name - 'model_lasso'):

Cross Validation Score : $0.858 + / - 0.046$

Model-02 (Name - 'model_ridge'):

Cross Validation Score : $0.851 \ + / - \ 0.043$

Model-03(Name - 'model_elastic_net'):

Cross Validation Score : $0.858 \ + / - \ 0.046$

Support Vector Machine Models:

Model-01 (Name - 'svm_li'):

Cross Validation Score : $0.668 \ + / - \ 0.060$

Model-02 (Name - 'svm_poly'):

Cross Validation Score : $0.723 \ + / - \ 0.030$

Model-03 (Name - 'svm_rbf'):

Cross Validation Score : $0.768 \ + / - \ 0.034$

Decision Tree Model:

Model-01(Name - 'tree_reg'):

Cross Validation Score: $0.815 \ + / - \ 0.091$

Ensemble Learning Models:

Random Forest(Name - 'rnd_forest'):

Cross Validation Score: $0.833 \ + / - \ 0.057$

AdaBoostRegressor(Name - 'ada_boost'):

Cross Validation Score: $0.912 \ + / - \ 0.061$

GradientBoostRegressor(Name - 'grd_boost'):

Cross Validation Score: $0.838 \ + / - \ 0.061$

Hyperparameter Tuning:

For Random Forest:

Grid Search(Name - 'grid_search_rnd_forest'):

Best Score: 0.9013

Random Search(Name - 'random_search_rnd_forest'):

Best Score: 0.8970

For Adaptive Boosting:

Grid Search(Name - 'grid_search_ada_boost'):

Best Score: 0.8512

Random Search(Name - 'random_search_ada_boost'):

Best Score: 0.8477

For Gradient Boosting:

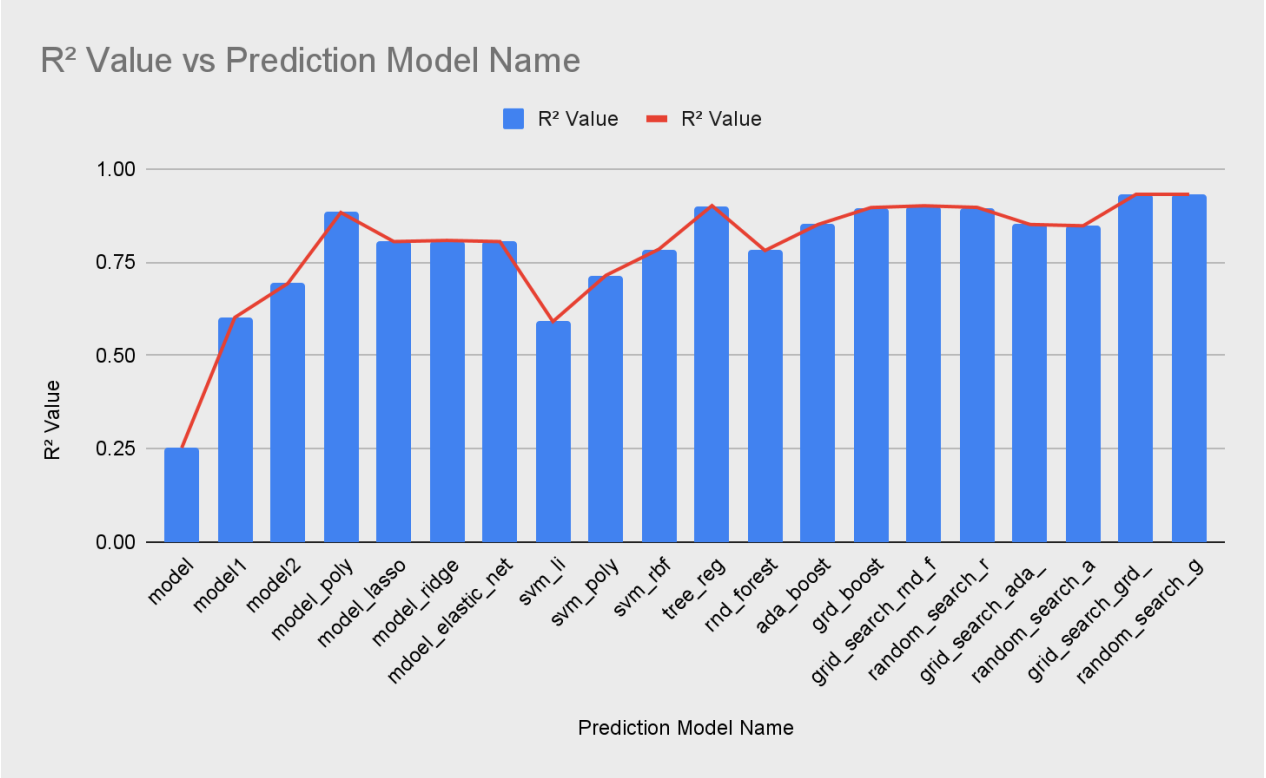
Grid Search(Name - 'grid_search_grd_boost'):

Best Score: 0.9322

Random Search(Name - 'random_search_grd_boost'):

Best Score: 0. 9322

Graph and table for the all the different models with their R^2 and Cross Validation Score:



S. No.	Prediction Model Name	R^2 Value	Cross Validation Score
1	model	0.2529	0.353 +/- 0.092
2	model1	0.6021	0.671 +/- 0.056
3	model2	0.6932	0.671 +/- 0.056
4	model_poly	0.8833	0.911 +/- 0.038
5	model_lasso	0.8054	0.858 +/- 0.046
6	model_ridge	0.8085	0.851 +/- 0.043
7	mdoel_elastic_net	0.8054	0.858 +/- 0.046
8	svm_li	0.5912	0.668 +/- 0.060
9	svm_poly	0.7149	0.723 +/- 0.030
10	svm_rbf	0.7851	0.768 +/- 0.034
11	tree_reg	0.9017	0.815 +/- 0.091
12	rnd_forest	0.7815	0.833 +/- 0.057

13	ada_boost	0.8512	0.912 +/- 0.061
14	grd_boost	0.8966	0.838 +/- 0.061
15	grid_search_rnd_forest	0.9013	0.9013
16	random_search_rnd_forest	0.8970	0.8970
17	grid_search_ada_boost	0.8512	0.8512
18	random_search_ada_boost	0.8477	0.8477
19	grid_search_grd_boost	0.9322	0.9322
20	random_search_grd_boost	0.9322	0.9322

5. Discussion

From the table mentioned above, we can clearly infer that polynomial regression models give the best prediction accuracy as compared to other models. Not only this, the regularizations also do not improve the accuracy for the model. In addition, it's clearly visible that the dataset is polynomially related to the target variable.

6. Conclusion

Based on the various models tried and all the approaches taken, we have been able to get a prediction model with more than 90% accuracy. Our baseline linear regression model has the accuracy of around 0.671with the error value of 0.05.
Other experimental models that have resulted in more than 90% accuracy are the polynomial model(‘model_poly), adaptive boosting model(ada_boost). Finally the grading boosting with the hyperparameter tuning using gradient search achieves the best accuracy of 0.93.
All these better performing models are almost $1.5\times$ better than the baseline linear regression model.

7. References

1. Akay,Mehmet. (2018). Optical Interconnection Network . UCI Machine Learning Repository. https://doi.org/10.24432/C5J60X.

2. Khan M. Iftekharuddin, Mohammad A. Karim,
Optical Interconnection Networks, Editor(s): Peter W. Hawkes, Advances in Imaging and Electron Physics, Elsevier, Volume 102, 1997, Pages 235-271, ISSN 1076-5670,
ISBN 9780120147441, https://doi.org/10.1016/S1076-5670(08)70124-1.
(https://www.sciencedirect.com/science/article/pii/S1076567008701241)